Performance Improvements on Intel® Architecture based Multiprocessor Workstations:

Multithreaded Applications using OpenMP

Intel®

September 2000 Intel Corporation

Table of Contents

1.0 About this white paper:

This white paper was written using resources from Kuck and Associates, an Intel Company (KAI), from Intel Multiprocessor Marketing, and from the Intel Microcomputer Software Laboratory. Special thanks goes to Henry Gabb of KAI and to Tim Mattson of Intel Corporation.

2.0 Abstract:

Analyst reports and industry research of market segments confirm that Intel® Architecture based workstation sales continue to outpace traditional workstation sales (International Data Corporation, Dataquest, April 2000). More and more Intel®-based workstations are being purchased with multiple processors, opening up opportunities for the Enterprise Information Services and Independent Software Vendors (ISVs) to deliver higher performing, multithreaded applications. In this paper, the scalability of several ISV applications is discussed in detailed benefit/cost terms. Each runs on an Intel® Architecture based multiprocessor workstation configured with the Microsoft* Windows NT* operating system. The scalability results of up to 98% were obtained on an Intel® 840 Chipset based workstation configured with two Intel® Pentium® III Xeon™ processors at 933 MHz with 256 KB Advanced Transfer Cache running on a 133 MHz system bus 512 MB Rambus* memory (see Appendix A).

These increasingly common Intel Architecture based multiprocessor workstations and servers have created greater demand for multithreaded applications, and are fueling renewed interest in parallel programming models and tools. This paper introduces OpenMP* as the dominant portable, scalable standard for easily adding parallelism to shared memory parallel (SMP) computer systems. First of all, the performance benefits obtained from using OpenMP to develop multithreaded applications on Intel Architecture based workstations are presented, and parallel computing concepts and terminology are defined. Next, an overview of the standard is provided, and examples of applications that benefit from OpenMP are reviewed. Lastly, examples and general guidelines for applying OpenMP are covered, and a reference guide to the compiler constructs is given in the appendix.

3.0 Areas of Opportunity for Intel® Architecture-based Multiprocessor Workstations

More and more applications are becoming available on Intel Architecture based workstations as ISVs gain insight into the market segment opportunity and take advantage of cost-effective development tools. Below is a sampling of opportunity classifications that ISVs are addressing as they port and develop applications that take advantage of parallelism on Intel Architecture based workstations.

3.1. Supercomputer Applications:

A range of such applications runs well now on Intel Architecture based multiprocessor workstations. One such example is Oxford Molecular's DGauss* for computational chemists.

3.2. Workstation Clusters:

Workstation clusters provide a low cost, scalable alternative to high-end parallel computers. Using Intel Architecture based multiprocessor workstations adds an additional level of performance to basic cluster computers. If the SMP nodes are connected using advanced clustering technologies like InfiniBand* (http://www.infinibandta.org) performance is further accelerated. In a cluster of SMP workstations, parallelism is exploited at two levels. First, work is distributed among the SMP nodes using message passing. Next, within each node, multithreading speeds the processing of local data. It has been shown that the combination of

message passing with multithreading often gives superior performance to either method alone (see Paper 1 in the Reference section).

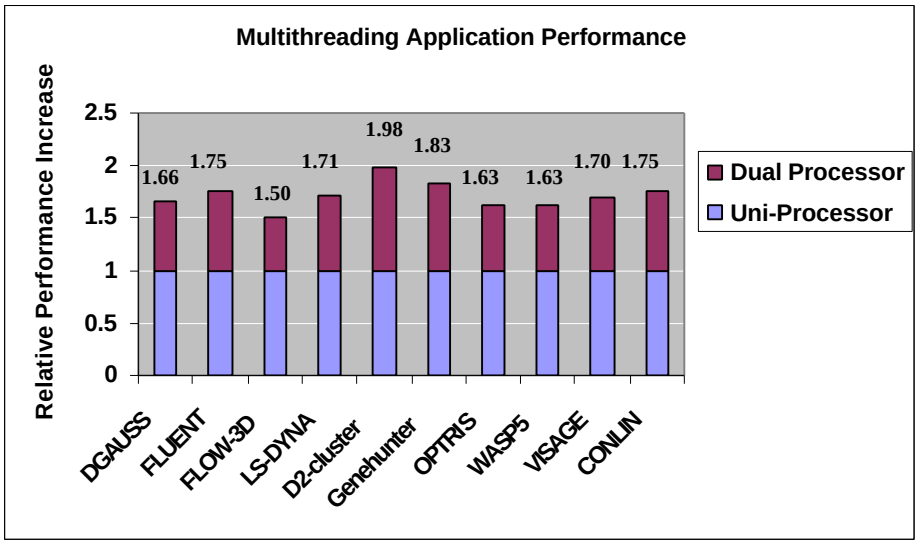
3.3. Legacy Desktop and Server Applications:

Applications historically associated with RISC-based UNIX* workstations and servers are now being ported to and developed for Intel Architecture based systems of equivalent or higher performance. The broadly available Intel Architecture workstations enhance Total Benefits of Ownership as compared to the Total Cost of Ownership by providing new software and services and a lower cost than the traditional platforms. VIPS VISAGE* provides an example of this type for petroleum and mining engineering.

3.4. New Market Segments:

As ISVs bring applications to Intel Architecture based workstations, they can develop new classes of users for their products. In this way, KBS2's LS-DYNA*, a widely used auto-crash code, has found applications in simulating drop testing of electronics gear.

Figure 1. Bar chart summarizing the performance multiplier available from dual-processor workstations based on the Intel Pentium lll Xeon processor 933MHz.



Source: Results from tests performed by Intel. Please refer to the Performance Report Notice at the end of this document. For system configuration information, see Benchmark System Configurations at the end of this document. See section 7.0 for details on how the work was done.

4.0 Introduction to OpenMP

OpenMP is the leading technology for expressing parallelism on shared memory architectures. OpenMP is an evolving model created by and maintained by the OpenMP Architecture Review Board (ARB). The ARB includes active participants in the world of shared memory computing such as Digital Equipment-now Compaq, HP, IBM, Intel, SGI, Sun Microsystems, KAI and scientists from the U.S. Department of Energy's ASCI Program. Read on further and gain exciting insights into OpenMP and parallelism.

5.0 Goals of Parallelism

The goal of parallel processing is to perform more work in less time. This can be accomplished either by increasing the raw processing speed of the computer, or by making use of exploitable concurrency within the task to perform multiple things at the same time. Parallel scalability can be used to decrease the runtime of the same job, or to perform more work in the same amount of time. Shared memory multiprocessing systems can deliver significant aggregate performance gains through the use of multitasking, multithreading and clustering, which are enabled by technologies such as OpenMP, Thread Libraries (Pthreads), and Message Passing Interface (MPI).

5.1 Multitasking:

One form of exploitable concurrency is called "multitasking". Multitasking simply means running more than one application at the same time. While most people using workstations do this every day, with multiprocessor systems, applications no longer need to compete for time on a single processor. Having two processors available can eliminate unnecessary wait time hence allow the user the opportunity to be more productive. Although each job individually may not run any faster, concurrently executing applications can be run in parallel across multiple processors, resulting in higher total performance.

5.2 Multithreading:

The most significant performance gains come from properly multithreaded applications. Multithreading allows application developers to divide the workload among multiple processors in parallel in order to either make the individual application run faster, or to increase the size of problem that can be solved. This capability gives developers greater flexibility and efficiency in the allocation of system resources. Applications specifically written to take advantage of multithreading on Intel Architecture based multiprocessing workstations can provide the significant performance gains that help companies maintain their competitive edge. Multithreading can be achieved using industry tools in the market segment today. Multithreaded applications are currently available in areas ranging from mechanical design, chemistry, and fluid dynamics to image rendering, and advanced simulations.

5.3 Message Passing Interface (MPI):

MPI is the dominant standard for developing parallel applications on distributed memory systems, and is used to process parallel tasks across distributed compute nodes. MPI is a standard library that enables parallel programming of clusters of computers. The standard supports development in C/C++ and Fortran. MPI programs are groups of independent processes or services with multiple code segments operating on separate data blocks (like Multiple Instruction, Multiple Data (MIMD)), with inter-process communication and coordination handled through the use of calls to MPI communication primitives. The message passing paradigm benefits from the ready availability of and the price/performance benefits of Intel Architecture based high performance networking devices (interconnects) and processors.

6.0 OpenMP

6.1 What is OpenMP?

OpenMP is a preferred technology for developing parallel applications on shared memory architectures from many leading vendors. OpenMP provides a simple, portable, high-level approach to add multithreading to new and existing applications, and has been widely endorsed, and adopted by leading computer manufacturers, operating system developers, and independent software vendors (ISVs). OpenMP grew out of the computer

industry's need for a portable, scalable standard that provided programmers with a simple and flexible interface to exploit shared memory multiprocessing on platforms ranging from the desktop to the supercomputer. The standard was jointly defined by leading computer hardware, software and tool manufacturers such as SGI, KAI, Intel, DEC (now Compaq), and IBM, and supports multi-platform shared-memory parallel programming in C/C++ and Fortran on all architectures, including Unix/Linux* and Windows NT based platforms. The standard also substantially extends the C/C++ and FORTRAN languages by providing compiler directives, library functions, and environment variables. OpenMP compiler directives serve to extend the base language sequential programming model and provide constructs for single program multiple data (SPMD), worksharing, and synchronization. The model also provides support for data sharing and privatization. Developers using OpenMP can benefit from multi-platform / multi-OS support, incremental parallelization of sequential code, a single source code solution, ease of use vs. thread libraries or message passing, and verifiable correctness of parallel programs. OpenMP tools can further assist application development by identifying time-consuming program areas, by automatically generating OpenMP directives, and by debugging and verifying parallel/sequential results.

6.2 OpenMP Comparisons and Contrasts

Though the present paper describes applications that have been threaded using OpenMP, other thread libraries like POSIX* threads, Solaris* threads, and Win32* threads are also available. Libraries are more robust than directive-based multithreading. The programmer controls thread creation explicitly so threads may be spawned at any level of the program. Therefore, multiple levels of threading (i.e., nested parallelism) are possible. This is not yet possible with OpenMP. However, thread libraries are low-level compared to OpenMP directives. Also, thread libraries usually require significant source code modifications, so the original serial program is not maintained.

6.3 OpenMP, MPI and Thread Libraries

OpenMP and MPI represent the two popular technologies for developing parallel applications. OpenMP is for shared memory programming and can be thought of as an API to exploit parallelism within the box (node). MPI is for distributed memory computing to exploit parallelism between the boxes (nodes). OpenMP is a multithreading model, which uses compiler directives to explicitly define areas of parallelism on multiprocessor, shared memory nodes (systems). Parallel regions consist of multiple threads within the same process sharing a common data area and heap. On the other hand, MPI is a library-based standard, which enables parallel processing across distributed memory computer systems. In MPI, programs consist of multiple processes with multiple code segments operating on separate data. Both OpenMP and MPI support the C/C++ and Fortran languages, and in some cases may be combined within the same application - MPI to enable parallelism between nodes, and OpenMP to exploit parallelism within a multiprocessor node.

Table –1 Comparison of Parallelism Models

The following table presents a high level comparison of the feature sets of three well recognized parallel computing standards.

Table 1. Table comparing the OpenMP, Pthreads, and MPI parallel programming methods

Feature	OpenMP	Pthreads	MPI
Portable	✓	**Vendor-dependent**	✓
Scalable	✓	✓	✓
Supports data parallelism	✓	✓	✓
Standard C/C++ API	✓	✓	✓
Standard Fortran API	✓		✓
Distributed-memory parallelism			✓
Incremental parallelism	✓		
High level	✓		
Serial code intact	✓		
Verifiable correctness	✓		
Development economics	**Low**	**High**	**High**

NOTE: ✓ represents "As designed"

- **Portable**: Portability determines whether a program can be moved to a new system (compiler and/or hardware) without costly modification. OpenMP is primarily a set of compiler directives designed to express parallelism on shared memory multiprocessors (SMP). OpenMP implementations are available for every SMP. Pthreads is a collection of library functions designed to express task-level concurrency. Though it's a POSIX standard library, vendor implementations often deviate from the standard in subtle ways, making porting difficult. MPI is another library that allows programmers to explicitly code the communication between parallel processes. It is designed for distributed memory architectures, in which each compute node has its own memory and cannot directly access another node's memory. MPI, however, can be used on either shared or distributed memory parallel architectures. In that sense, it's more portable than OpenMP or Pthreads.

- **Scalable**: All three methods can be used to develop programs that provide continuous performance improvements (i.e., scale) as more processors are added to the computation.

- **Supports data parallelism**: The data parallel programming model exploits the inherent parallelism of applying the same operation repeatedly to different data. Loop-level parallelism is a good example. OpenMP provides a robust set of directives to express several types of loop-level parallelism. Though possible, expressing data parallelism with Pthreads and MPI is more difficult.

- **Standard Fortran, C/C++ API**: self-explanatory

- **Distributed-memory parallelism**: self-explanatory

- **Incremental parallelism**: With OpenMP it is easy to parallelize distinct regions of code without affecting the rest of the application, allowing the programmer to gradually increase the level of parallelism. More importantly, non-OpenMP compilers ignore OpenMP directives so the original serial code is retained. Parallel libraries require a permanent commitment. Pthreads and MPI both require significant modification of the original code to express even simple parallelism.

- **High level**: see incremental parallelism

- **Serial code intact**: see incremental parallelism

- **Verifiable correctness**: Because the original serial program is left intact by OpenMP directives, it is possible to verify the correctness of parallel execution relative to serial execution down to the level of individual memory accesses. This is not possible with parallel libraries.

- **Development economics**: OpenMP is superior to Pthreads and MPI in nearly every respect. Incremental parallelism is possible, so is verifiable correctness, both of which make development and debugging easier. OpenMP is high level, which makes parallel development easier. Data parallelism can be expressed more easily with OpenMP directives than with calls to parallel libraries.

Table 1 summarizes the advantages and disadvantages of the most common parallel programming methods. POSIX* threads, or Pthreads, is a standard thread library that is more robust, but also more complicated, than OpenMP. It gives the programmer greater control over thread creation and manipulation. Its greatest advantage over OpenMP is that multiple levels of threading (i.e., nested parallelism) are possible. Multithreading is not the only parallel model. Message passing allows applications to be executed on multiple processors with physically distributed memory (e.g., workstation clusters). Data is exchanged between processors by way of messages that must be explicitly coded. The industry standard Message Passing Interface (MPI) is the most widely used library for parallel programming on distributed memory systems.

Documentation of the parallel model of the application is important so that new functionality can be compared against the model as it is being implemented. Because documentation tends to fall behind the actual implementation, the best parallel programming methods should be self-documenting. OpenMP provides significant advantages in this regard. Using OpenMP, this incremental approach to parallelism is also straightforward for new parallel application development. This approach creates savings as well as opportunities to increase time to market.

6.4 OpenMP Benefits

The benefits of using OpenMP include:

- Getting greater performance from multithreaded applications running on multiprocessor workstations
- Faster time to market since OpenMP uses a simplified programming model which allows incremental loop-level parallelism of code
- Reduced support costs given that OpenMP allows multi-platform development with a single source code
- Users gain the benefits of verifiable correctness, meaning that multithreaded versions of code can be shown to produce the same result as the sequential version.
- Every application in wide use is under continual development on two fronts: more meaningful simulations are being developed and faster software implementations are being introduced. Especially for parallel processing, these two development threads are sometimes in short-term conflict. However, the

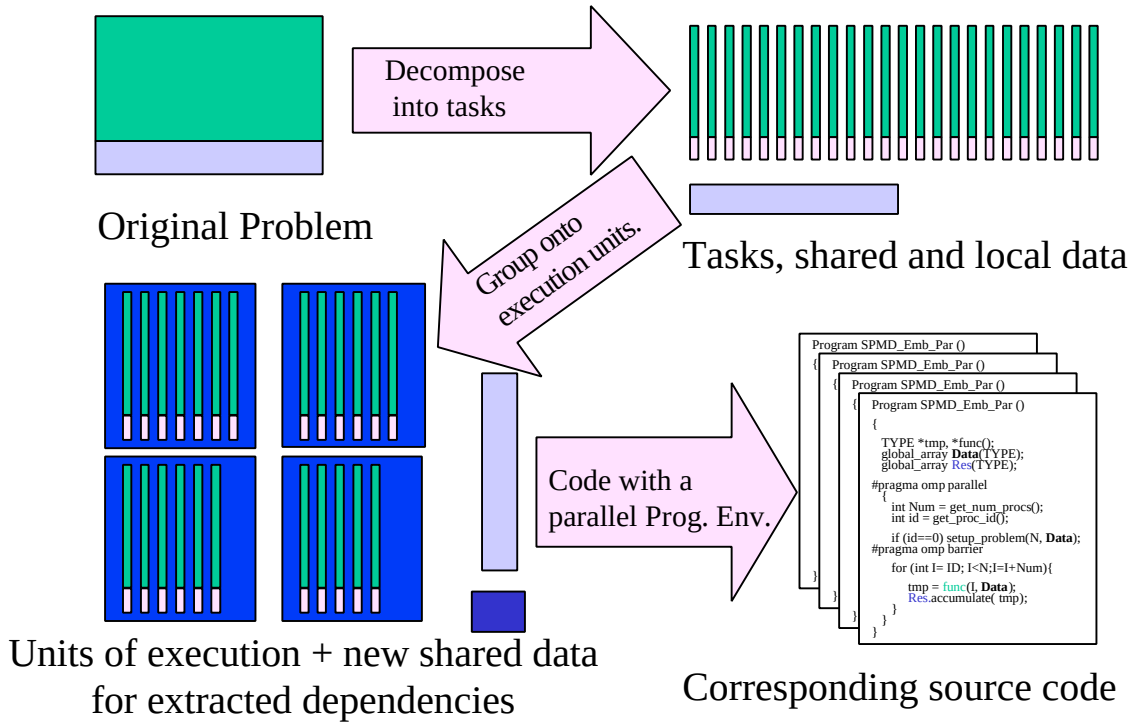
use of powerful parallel software engineering techniques on widely used, increasingly powerful Intel Architecture systems will lead to a broader understanding and realization of increased benefit and reduced costs in delivering parallel application software.

#Disclaimer: These benefits depend on individual system configurations and circumstances, and may not always be available.

6.5 OpenMP Programming Model

OpenMP uses a fork-join model of parallel execution. Application execution begins as a single master thread, and continues until a user-defined parallel construct is encountered marking the start of a parallel region. At this point, the master thread creates a team (fork) of threads and becomes master of the team. Within a parallel region, either multiple threads can execute the same block of code, with code statements used within the block to ensure that the threads work on different data, or work-sharing constructs may be used to execute different code sections in parallel. Multithreaded parallel execution then resumes until an end parallel construct is encountered marking the end of the parallel region and a return to serial processing (join). An implied barrier is assumed at this point. The barrier is a synchronization construct, which ensures that all threads complete before execution continues again as a single master thread. In this way, all threads will have reached a known state after the barrier region. Additional threads, which were created during the parallel region, either sleep or spin. Using this approach, parallelism may be added incrementally by parallelizing time consuming loops, and by splitting the work among threads.

6.6 Methods of Creating OpenMP Programs



For developing new applications, the process involves analyzing the original problem, decomposing it into tasks using both shared and local data, resolving the data dependencies, and then regrouping the tasks into execution units which are coded using a parallel programming environment. For existing applications, an analysis of where the program spends most of its time should be performed. An application profiling tool such as the Intel®

VTune™ Performance Analyzer can be used. A parallel library can be used for applications that perform most of their work while calling well-defined library routines. For other types of applications, a parallel API can be used.

6.7 OpenMP Enablers

6.7.1 InfiniBand

Workstation clusters provide a low-cost, scalable alternative to high-end parallel computers. Using Intel Architecture based multiprocessor workstations adds an additional level of performance to commodity cluster computers. If the SMP nodes are connected using advanced clustering technologies like InfiniBand, performance is further accelerated. In a cluster of SMP workstations, parallelism is exploited at two levels. Work is distributed among the SMP nodes using message passing, MPI. Within each node, multithreading speeds the processing of local data. It has been shown that the combination of message passing with multithreading often gives superior performance to either method alone (see Paper 1 in the Reference section).

6.7.2 OpenMP Tool Enablers

Increasing numbers of multithreaded applications have become available on Intel Architecture based multiprocessor workstations, as ISVs understand the market segment opportunity and performance benefits of developing with OpenMP using OpenMP Tools. Moreover, parallel software development tools are currently available to ease the migration to OpenMP on Intel Architecture and to accelerate and simplify the porting and development process. Current tools are also able to perform correctness verification of the parallel program, performance profiling, and automatic parallelization of code. At the same time, application users can easily scale the number of processors to match their performance requirements.

Table of Several OpenMP Tools

Tool	Description	C/C++ or Fortran Supported
VTune™ Intel Performance Analyzer	Performance Analyzer for developing optimized applications that take advantage of the latest Intel processor technology.	Fortran and C/C++
KAP/Pro Toolset*	A toolset for OpenMP that combines a complete OpenMP implementation with unique supporting development tools to make it easy to add parallel threading to existing software.	Fortran and C/C++
Portland Group Toolset	A set of parallel Fortran, C and C++ compilers and tools for Intel processor-based Linux, NT or Solaris86 workstations or servers.	Fortran and C/C++
RogueWave Threads.H++	A product to help quickly multithread existing applications and create new multithreaded applications from scratch.	C++

6.8 Parallel Software Engineering Tools

Parallel software engineering tools are now available to express parallelism in a high level form using OpenMP and to analyze new code as it is being developed. These tools allow the programmer to determine quickly when flaws develop in the parallel code, and to discover when performance starts to degrade. Currently it is feasible to incorporate correctness and performance tests into a regression suite for OpenMP applications.

Kuck and Associates, an Intel company (KAI) (http://www. kai.com) offers the KAP/Pro Toolset for Windows NT, which was used in all of the work discussed. The KAP/Pro Toolset contains:

- **Guide**, which processes OpenMP directives for Microsoft Windows or UNIX-based application software
- **GuideView, *** which provides a detailed profile of the components of parallel performance for each part of the program
- **Assure**, which verifies the correctness of a parallel program relative to the serial version of the program; and
- **Directive** translators for converting OEM-specific directives to OpenMP.

KAP is also available for autoparallelizing those parts of an application that do not require manual attention.

KAP/Pro Toolset Specifications

The KAP/Pro Toolset has been used for parallel software engineering on many ISV applications, including all of the examples in this paper. In some of these examples, legacy parallelism was built into applications using proprietary directives. These were translated to OpenMP using KAP/Pro. The complete update of these applications was done rather quickly (measured in days) by using KAP/Pro's **Guide**. Debugging parallel programs is frequently difficult. Using print statements or multiple instances of standard debuggers can be cumbersome, although multithreaded debuggers are making their way to Windows NT. Using KAP/Pro's Assure tool to locate bugs automatically accelerated debugging in the examples in this paper. Application developers build their programs with the **Assure** instrumenter (in a process like a normal compile), which instruments the serial program to emulate a parallel processor. The instrumented application is then run with typical data sets to produce a database of suspicious events that occurred in the **Assure** emulation. **AssureView*** provides a developer interface for fixing initialization, final value, privatization, and data race bugs. After debugging, the parallelized applications in this paper were tuned for performance. In tuning parallel applications, it is important to note that performance gains depend directly on the fraction of the application that can be run in parallel. One useful way to estimate this is with Amdahl's law:

$$\textit{Performance Improvement} = T / [f * (T/P) + (1 - f) * T]$$

Where:

T = Scalar execution time
f = Percentage of sequential time that runs in parallel on P processors
P = Number of processors

For example, if 75% of an application's sequential time can be run in parallel, the potential performance multiplier on a dual processor Intel Architecture workstation over a uniprocessor workstation is 1.6. Stated differently, this says an application that runs in 1 hour on a dual processor workstation would take 1.6 hours (1 hour, 36 minutes) to run on a uniprocessor workstation.

With this understanding, software engineers can measure the actual performance and begin tuning. In the examples of this paper, the **GuideView** parallel performance analysis tool was used to analyze performance derived from the use of OpenMP directives.

GuideView provides information on system performance bottlenecks such as:

- Synchronization time in locks and critical sections
- Loop imbalance due to the scheduling algorithm chosen
- Sequential time in code sections remaining as parallelization candidates

When bottlenecks were identified, modifying the directives and/or modifying the application source code addressed opportunities for performance improvement.

6.9 Applying OpenMP

6.9.1 Typical Use (When to use or not to use - what applications can benefit)

OpenMP is typically used to speedup applications by parallelizing time consuming program loops, and by splitting the work among several threads. OpenMP provides a simple, portable, standardized API for shared memory computer systems, which can readily be used to thread both new and legacy code. For existing applications, an analysis tool such as the Intel VTune Performance analyzer may be used to create a profile of time-consuming program areas (See the Reference section for a web link). Applications, which spend most of their time in well-defined math library routines, can add parallel processing by linking in a parallel math library such as the Intel Math Kernel Library (MKL). For other categories of applications, OpenMP should be considered for programs, which spend a considerable amount of time processing independent blocks of data. Typically, this can be implemented as either multiple copies of a single program executing on multiple threads and applied to a subdivided data set (Single program, multiple data (SPMD)) or multiple different code sections executing on multiple threads processing multiple independent blocks of data (Multiple program, multiple data (MPMD)). Care should be taken to ensure that the size of the independent job is large enough to justify the overhead of running the job in parallel.

6.9.2 Constructs and Examples

OpenMP incorporates five general categories of constructs that take the form of either compiler directives or pragmas. These are parallel regions, worksharing, synchronization, data environment, and run-time functions/environment variables. Parallel regions are sections of code where the main (master) thread splits into a team of threads in order to perform work in parallel. If a worksharing construct is not employed, the same section of code is executed by all of the threads. Typically, the code has decision points, which determine the data areas that are processed by each thread. Worksharing constructs allow different code sections (task level parallelism) to execute in parallel. Synchronization constructs provide a means of coordinating the processing of work by the threads. Some of these constructs allow only one thread to access a section of code at a time (master, critical, atomic), while others ensure that the threads have all reached a common state (flush, barrier). The data environment constructs provide directives and clauses, which control the scope of variables during the execution of parallel regions. An example of this would be the threadprivate directive, which makes global variables local to a thread. Finally, run-time functions and environment variables provide information on or set options within the run-time environment. These include such things as the number of threads, thread number, and whether or not nested parallel regions or dynamic adjustment of thread number are allowed.

6.9.3 Code Examples

Loop-Level Parallelism	Task-Level Parallelism	Parallelism Without Worksharing Constructs
This example illustrates simple loop-level parallelism for an elemental array multiplication. The work contained in the loop is automatically divided among the threads.	This example shows how OpenMP can be used to express Task-level parallelism, in which each thread executes a particular section of code. If the amount of work in each section is significantly different, the load imbalance will adversely affect scalability.	It is also possible to write OpenMP code without using worksharing constructs (e.g., do, sections, single). Done correctly, this coding style can give very good performance and scalability. However, it is complicated and time-consuming. Start the parallel region as high as possible in the program and explicitly compute the domain of each thread. Fortran 90 array syntax is well suited to this style of OpenMP programming. (For simplicity, assume that the number of threads and array elements divide evenly in the example code.)

real a(N), b(N), c(N)
!$omp parallel do
do i = 1, N
a(i) = b(i) * c(i)
enddo
	real a(N), b(N), c(N), d(N)
	!$omp parallel sections
	!$omp section
	do i = 1, N
	a(i) = 0.0
	enddo
	call laplace(a)
	!$omp section
	do i = 1, N
	b(i) = 0.0
	enddo
	call eigen(b)
	!$omp section
	call read_next_data(c)
	data_ready = .true.
	!$omp section
	call write_previous_results(d)
	!$omp end parallel sections
		integer :: i1, i2, my_id, nt
		integer, parameter :: N = 1000
		real, dimension(N) :: a, b, c
		
		!$omp parallel default(shared) private(i1,i2, my_id)
		nt = omp_get_num_threads()
		my_id = omp_get_thread_num()
		
		! Each thread determines the lower and upper bounds of its array section
		i1 = my_id * N / nt + 1
		i2 = (my_id + 1) * N / nt
		
		! Each thread calls random to initialize its array section
		call random(b(i1:i2))
		call random(c(i1:i2))
		
		! Each thread performs its portion of the array multiplication
		a(i1:i2) = b(i1:i2) * c(i1:i2)
		
		! Master threads writes result
		if(my_id == 0) then
		write(*,*) a
		endif
		
		if(mod(my_id, 2) == 0) then ! Even numbered threads do one thing
		call do_one_thing()
		else ! Odd numbered threads do something else
		call do_another_thing()
		endif
		!$omp end parallel

7.0 Multiprocessing Application Threading Benchmark Details

The Multiprocessing Application Threading Benchmarks are presented below. In each case study, applications are described, with a sketch of how the parallelism work was done, and the performance achieved in each case. All test cases were run using realistic data sets in cooperation with the ISV developers in the application areas of:

- Computational Chemistry, Computational Fluid Dynamics, Crashworthiness Analysis
- Genetics, Multivariate Optimization
- Metal Stamping, Petroleum, Environmental, Civil and Mining Engineering

There are two ways of interpreting the performance multipliers. Let's use the FLUENT* simulation as an example. The simplest interpretation is to say that in the time it took the single processor workstation to do one unit of work, the dual processor workstation did 1.75 times as much work (see Table 2). The second way to look at it is to say that the dual-processor workstation completed a given amount of work in 75% of the time it took the single processor workstation to do the same thing. This is a time saving of 387 seconds (see Table 2). The table included here provides more detail on each of the tests in the bar chart (Figure 1).

Table 2. Table summarizing the multithreaded performance multiplier based on the Intel Pentium lll Xeon dual-933 MHz processor workstation.

Source: Tests performed by Kuck and Associates (an Intel company). Refer to the Performance Report Notice at the end of this document. For system configuration, see Appendix A at the end of this document.

Application	Number of threads		Speedup*
	1	2	
DGAUSS	58 min.	35 min.	1.66
FLUENT	906 sec.	519 sec.	1.75
FLOW-3D	241 min.	161 min.	1.50
LS-DYNA	77 min.	45 min.	1.71
D2-cluster	527 min.	266 min.	1.98
Genehunter	84 sec.	46 sec.	1.83
OPTRIS	31 min.	19 min.	1.63
WASP 5	163 min.	100 min.	1.63
VISAGE	34 min.	20 min.	1.70
CONLIN	56 min.	32 min.	1.75
		Average Gain	**1.71**

*The performance results presented here are total runtime (including I/O and serial regions), not just the runtime of the multithreaded computations. Parallel speedup is higher when only the multithreaded regions are considered.

7.1 Computational Chemistry

Oxford Molecular Group, DGauss*

Oxford Molecular Group (http://www.oxmol.com) is an international supplier of software and services for discovery research with computational chemistry. Their software is installed at all the major chemical and pharmaceutical companies. DGauss, now available as an add-on to the CAChe* (Fujitsu Systems Business of America, http://www.cache.fujitsu.com) suite of molecular modeling software, is an Oxford Molecular application originally developed for Cray supercomputers. Intel Architecture multiprocessor workstations configured with Microsoft Window NT now enable chemists to effectively reduce time to solution in the development of new chemicals and drugs. The portability provided by the KAP/Pro Toolset allowed the code to be moved easily from UNIX supercomputers and workstations to Intel Architecture systems. (For more details see Paper 2 in the Reference section.)

How the work was done

Guide, part of the KAP/Pro Toolset, enables software developed to run on a Cray* to migrate easily to Intel systems running Windows NT by translating non-portable parallel compiler directives to OpenMP. With OpenMP, a developer can easily increase the number of processors that can be effectively applied to parallel computations. GuideView was used to find and tune performance bottlenecks (see Paper 2). Oxford Molecular Group used Assure, a tool for debugging and Q/A to bring new functionality to the market segment quickly on Intel Architecture systems.

Performance Demonstration

A molecule from the silicalite series, a silicon-containing zeolite subcluster, was modeled. The sequential computation time was approximately one hour. Multiprocessing decreased the runtime to only 35 minutes – a speedup of 66%. Table 2 shows the performance benefits of KAP/Pro OpenMP on Intel Architecture multiprocessor workstations.

7.2 Computational Fluid Dynamics

Computational fluid dynamics (CFD) is a computationally intensive but extremely important engineering method. CFD is used in a broad range of engineering analyses from air flow over aircraft wings to water flow through pipes to just about any design problem involving fluid flow. Two industrial CFD applications that have been multithreaded to take advantage of Intel SMP systems are described in the present paper.

Fluent Inc., FLUENT*

Fluent Inc. (http://www.fluent.com) is a leading ISV developing CFD software. FLUENT, one of their key applications, has run successfully on UNIX-based multiprocessors for several years and has also been available on single processor Intel Architecture multiprocessing workstations configured with Microsoft Windows NT for several years. Fluent Inc.'s cost of supporting parallelism on multiple platforms is managed by the KAP/Pro Toolset, which provides portability between RISC/UNIX and Intel Architecture SMP systems. Managing this cost is key because of the opportunity cost of each dollar spent porting software, which could have been spent developing new features in rapidly growing the CFD market segment.

How the work was done

The application developers had already introduced RISC/UNIX-vendor directives in FLUENT. The non-portable, parallel directives were replace with OpenMP. GuideView allowed the developers of FLUENT to analyze performance on Intel SMP systems compared to RISC/UNIX platforms. The KAP/Pro Toolset allowed FLUENT

developers to shift to Intel Architecture multiprocessing workstations rapidly and make side-by-side performance comparisons, significantly reducing time to market. The total porting and tuning effort invested by KAI and Fluent Inc. was only a few weeks.

Performance demonstration
A swirling turbulent flow in a curved duct with a sudden expansion inlet was simulated to measure the performance improvement from multithreading. The test system used approximately 200,000 cells. Table 2 illustrates that dual-processor Intel Architecture SMP workstations provide a substantial performance increase over a single processor workstation.

Flow Science Inc., FLOW-3D*

Flow Science Inc. (www.flow3d.com), another industry leader in CFD, is the maker of FLOW-3D, a general-purpose fluid modeling package. It is used primarily to model free surface flows or confined flows. However, FLOW-3D includes a variety of special features that enable the user to study acoustic waves, cavitation, the solidification and shrinkage of metals, flow through porous media, surface tension, wall adhesion, etc. The FLOW-3D preprocessor can automatically create structured grids from solid models. Structures can also be imported from CAD programs.

How the work was done
Though the profile was relatively flat, with no single routine consuming more than 25% of the total runtime, there were several good targets for multithreading, the most obvious being the loop over active cells. Modifications to the source code were necessary, however. For routines called inside parallel regions, variable passing via global data was changed to explicit argument passing. A technique called wavefronting was used to parallelize the loop over active cells. Wavefronting parallelizes the algorithm in diagonal waves over the original iteration space. This required additional arrays to store wavefront data. The FLOW-3D multithreading effort was an Intel QuickStart project that took approximately one month. Some parts of the computational kernel still require multithreading.

Performance demonstration
A high pressure casting simulation was used to test multithreaded performance. Molten aluminum is injected into an alternator housing to compute filling patterns to determine if, and where, air bubbles get trapped. Air bubbles in the mold cause defects in the alternator housing. Understanding the filling process allows design engineers to adjust gating and/or fill-rate to eliminate trapped air. Simulations of this complexity take several hours. Adding an additional thread to this computation cuts runtime nearly in half.

7.3 Crashworthiness Analysis

KBS2, LS-DYNA

LS-DYNA (http://www.kbs2.com) is a robust, multifaceted MCAE analysis tool that is used extensively by both commercial and educational/research engineering clients. Applications of LS-DYNA have expanded, in part due to its availability on Intel Architecture multiprocessing workstations configured with Microsoft Windows NT, to include aerospace, bio-medical and electrical component drops testing applications, as well as automobile crashworthiness and metal forming. LS-DYNA routinely performs complex analyses with relative ease, due to its user-friendly Intel Architecture multi-processing workstation operational environment, command line mode, and on-line manuals. LS-DYNA is an example of the expanding market segment available to ISVs as new applications arise.

How the work was done

The Intel Architecture multiprocessing workstations parallelization effort of LS-DYNA was performed under an IBM/KAI QuickStart program, with KBS2, which supports and distributes the Intel Architecture multiprocessing workstation version of LS-DYNA. Both Assure and GuideView have effectively enhanced the benefits in producing and maintaining LS-DYNA. (For more information about parallel LS-DYNA with KAP/Pro, see Paper 3 in the Reference section).

Performance Demonstration

Performance tests of LS-DYNA on an Intel Pentium III Xeon processor, 933 MHz, based workstation challenges the high-end UNIX/RISC workstation capabilities. For the test system analyzed in this paper, a small car crashing into a pole, runtime was reduced from 77 minutes to 45 minutes. For engineers whose design work includes a number of "run LS-DYNA—think—rerun LS-DYNA" cycles, it becomes possible to move from one or two cycles per day, to seven or eight, depending on processor speeds.

7.4 Genetics

South African National Bioinformatics Institute, d2-cluster

Sequence-tagged-sites, discovered during the Human Genome Project, are small stretches of DNA that occur only once in an entire genome. Expressed sequence tags (EST) from particular genes are used to map the enormous amount of genetic data made available by modern sequencing techniques. EST's are particularly useful for finding important genes. D2-cluster, part of the STACKPACK* clustering system provided by the South African National Bioinformatics Institute (SANBI, http://www.sanbi.ac.za), is designed to rapidly generate EST clusters. The SANBI and the University of Texas Health Science Center recently used STACKPACK to locate the gene for retinitis pigmentosa, a common form of inherited blindness.

How the work was done

Database searches illustrate an important concept in multithreaded computing – granularity, the division of work between threads. To use a commonplace example, four mechanics, each changing a single tire, is probably faster than four mechanics working together on each tire. Likewise, when scanning a genetic database (or any other database), multiple threads conducting independent searches is better than having the threads collaborate on each sequence comparison. Therefore, the main loop over the database records was the best target for multithreading. However, changes to the original source code were necessary to enable multithreaded searching. First, it was necessary to change the main search loop from a while-loop to a for-loop. (OpenMP does not define parallel while-loops.) Second, synchronization was necessary to merge the search results of each thread. Because each thread dynamically allocates and frees memory during sequence comparisons, a thread-safe malloc function was essential. This is provided in the Guide library. Parallel assurance testing was done with Assure. This work was done in a few days as an Intel QuickStart project.

Performance demonstration

The database searches performed by d2-cluster are predominantly parallel. Each thread scans the database independently, interacting with other threads only to merge data. Therefore, runtime should decrease linearly with the number of threads. Synchronization overhead will eventually degrade parallel efficiency. On the Intel dual-Pentium III Xeon processor 933 MHz workstation, however, two threads halves the time to cluster the database of 688 EST's provided by the SANBI.

Whitehead Institute, Genehunter*

Genehunter is a genetics program that analyzes a set of DNA assays from members of a family tree. The MIT Whitehead Institute distributes Genehunter (http://www.wi.mit.edu). Genehunter attempts to identify the gene most likely to cause a disease based on its presence in certain members of the family and not in others (see Paper 4 in the Reference section). Linkage analysis methods, like Genehunter, have success-fully located the genes underlying many diseases such as muscular dystrophy, cystic fibrosis, and Huntington's disease. In the future, these methods may be used to interpret enhanced sensitivity in individuals to common diseases such as influenza.

How the work was done

Earlier work to parallelize Genehunter using Pthreads had produced an initial version of a parallel code just as a new version of Genehunter was being released. Correctness and performance problems were observed, and KAI was invited in as a consultant. KAP/Pro was used to introduce OpenMP directives and then GuideView and Assure were used to develop a robust high-performance version of Genehunter. This work was done in about two weeks by KAI as an Intel-Compaq sponsored QuickStart project.

Performance Demonstration

The data set used in the run presented below consists of multi-generation families collected in the study of a common complex disease. The data is typical of the type of families collected for whole genome scans for susceptibility genes, common genes that make family members susceptible to the same disease.
The data set contains a mixture of small sib-ships in several three- and fourth-generation extended families. There are 564 individuals belonging to 72 different pedigrees (range 4-16 individuals/ pedigree). Individuals were genotyped for 17 multiallelic genetic markers with an average spacing of approximately 10 cM between markers. In this dataset, 359/564 individuals were genotyped with 80% success, and Genehunter testa1 dataset had an 83% performance boost (see Table 2).

7.5 Metal Stamping

Dynamic Software, Optris*

Dynamic Software (http://www.dynamic-software.com) develops and markets Optris. Auto manufacturers, toolmakers, steel suppliers, and others use Optris internationally to simulate the entire metal stamping process. In order to detect ruptures, wrinkles and other stamping problems before tool manufacturing and production begins, Optris is used to validate the results of a CAD design. As a virtual press, Optris replaces the try out phase on a real press. This reduces costs and lead times, and by optimizing designs it also increases the resulting product quality. Optris is based on an explicit finite element method. Incremental stamping simulation can now run on PCs with excellent Intel Architecture multi-processing workstations configured with Microsoft Windows NT performance.

How the work was done

Dynamic Software has used KAP/Pro Toolset on UNIX systems for some time. When their Intel Architecture multiprocessing workstations product needed more performance, they selected the known cost and benefits provided by Guide and GuideView to port and tune their OpenMP parallelism. Because adaptive meshing is done, OpenMP directives are very effective; message passing was quickly dismissed as an approach due to the dynamic behavior of adaptive meshing during these computations.

Performance Demonstration

For this paper, Optris was run on the top part of a Renault Twingo front suspension tower. The computation involved adaptive meshing with a model that began with 6,400 elements and ended with 12,500 elements. The blank undergoes an increase of nearly an order of magnitude in elements. A total of over 18,000 cycles were used in the simulation. Table 2 shows the KAP/Pro Intel Architecture multiprocessing workstation performance results for Optris, with a clear implication that scalability on more processors will be effective.

7.6 Petroleum, Environmental, Civil and Mining Engineering

Limno-Tech Inc. and the U.S. Environmental Protection Agency, WASP

The Water Quality Analysis Simulation Program (WASP) is used to model water quality issues such as oxygen dynamics and nutrients and eutrophication. It is also used to study various forms of contamination (e.g., bacterial, organic chemical, heavy metal) in water supplies. In addition, WASP can simulate the reactivity of chemicals and nutrients in surface water. WASP models these water quality issues as time-evolving processes. The U.S. Environmental Protection Agency (http://www.epa.gov/earth100/records/wasp.html) maintains and distributes the public version of WASP. The multithreaded version is available through Limno-Tech Inc. (www.limnotech.com/modeling/moddemos/quickstart.shtml).

How the work was done

Limno-Tech Inc. kindly provided WASP source code and input data. Fortunately, simulation length could be easily adjusted. An eight-day simulation was used to profile the serial code. This length was sufficient to exercise the computational kernel without I/O artifacts. Profiling revealed that about 75% of the total runtime was spent in three routines. Two of these routines required slight modification to facilitate multithreading. A key loop was restructured to avoid branching (i.e., goto statements) outside of the loop. Also, thread-local copies of some global variables were created to avoid undo synchronization, and hence, parallel overhead.

OpenMP directives were added to the three dominant routines and the KAP/Pro Toolset was used for debugging and tuning. A one-day simulation was used for parallel assurance testing with Assure. An 8-day simulation was used to optimize multithreaded performance with Guide and GuideView. This work was done in a few days as an Intel QuickStart project.

Performance demonstration

Performance measurements were done on an Intel 933 MHz Pentium III Xeon dual-processor workstation. A full-year WASP simulation of water quality in the Ohio River near Cincinnati, Ohio was used to measure the performance benefit of multithreading. (This type of simulation was used to support planning and design of sewage collection systems.) Multithreading the computational kernel reduced running times from approximately five hours to three hours. Not only does this allow longer simulations to be completed more quickly, it significantly improves the efficiency of WASP experiments that require multiple parameterization cycles.

VIPS Limited, The VISAGE* System

VIPS Limited (http://www.vips.co.uk) is a UK-based company that develops and markets The VISAGE System, an advanced and comprehensive finite element program for solving geomechanical, rock mechanical, environmental and petroleum engineering problems. The program has been used on supercomputers and high-end servers and recently was ported to and parallelized on Intel Architecture workstations. Using VISAGE, engineers can solve for and design mines, foundations, dams, bridges, and buildings. The same system can also be used for environmental studies in order to track down the flow of contaminants through porous media. Petroleum engineers can use the VISAGE software for the compaction and subsidence of black oil reservoirs,

design for the location of injection/production wells or study the initiation and propagation of fractures as VISAGE is the world's first three-phase, gas-oil-water-, stress-dependent reservoir simulator which includes thermal fracturing capabilities. The software can also be used in structural geology for determining the modern stress state between complex fault and fracture networks.

How the work was done
VIPS and KAI have been long-term partners on parallel UNIX systems, so the use of KAP/Pro Toolset under the KAI QuickStart program with Compaq was a natural next step. In a few weeks, VIPS was able to obtain high performance on their Intel Architecture system, and parallel VISAGE began shipping within a month.

Performance Demonstration
To demonstrate VISAGE performance, a foundation resting on a 3-D mass of stiff clay was modeled. The footing is loaded to represent the weight of the building and the deformations around the footing are calculated. The performance improvement due to multithreading is shown in Figure 1 and Table 2.

7.7 Multivariate Optimization

Samtech s.a., CONLIN*

Samtech s.a.® CONLIN (version 3.1) is a multivariate optimizer designed to solve large-scale problems (i.e., greater than 100,000 variables). Dr. Claude Fleury, a professor at the University of Liege and Senior Consultant to Samtech s.a, developed *CONLIN. The CONLIN solver relies on convex, separable approximations, which makes it particularly suited to problems with many design variables and few constraints. It is used primarily to optimize structures according to size, shape, and topological variables. It has also been applied to multidisciplinary optimization problems containing hydrodynamic and aerodynamic constraints. The CONLIN optimizer is used in the BOSS Quattro* application manager (http://www.samcef.com/BOSSquattro.html) and in Altair Engineering Inc.'s OptiStruct* optimization tool (http://www.altair.com).

How the work was done
Most of the computation is confined to two routines that handle topology optimization. Only slight modifications to the source code were necessary. For example, one of the parallel regions contained a series of inherently serial calculations that disrupted parallelism. These calculations were moved outside of the parallel region. It was also beneficial to create thread-local copies of some variables to avoid synchronization. Professor Fleury threaded CONLIN with technical assistance from Kuck and Associates.

Performance demonstration
The time to solve 600 different optimization problems, each with 100,000 design variables and between one and ten constraints, was used to measure serial and parallel performance. It took nearly an hour to complete the calculation using one processor, but only 32 minutes using two threads per optimization. The actual speedup due to OpenMP multithreading was 75%.

The results reported in this paper should be viewed as a snapshot of work in progress and should be considered as examples. As mentioned before, performance results will vary from application to application and from data set to data set.

8.0 Summary

Intel Architecture multiprocessor workstations are becoming commonplace and demand for multithreaded applications to run on them is growing. This paper shows that porting to Intel Architecture systems from UNIX systems is quite easy. Moreover, the performance of workstations based on the Intel Pentium III Xeon processor rivals many RISC processors. This makes Intel Architecture systems more attractive not only for application delivery but also for initial application development.

While there will be application-to-application variances, the scalability of systems from one to two processors on Intel Architecture systems running Microsoft Windows NT is comparable to systems based on other architectures. Two recent software events offer great opportunity and benefit to application developers using Intel Architecture systems. First, the introduction of KAP/Pro Toolset for Intel Architecture workstations running Microsoft*Windows*NT allows developers the same facilities on Intel Architecture / Microsoft Windows NT systems that had been available only on traditional systems. Second, the OpenMP standard for parallel programming allows developers new high-level approaches to parallel programming. OpenMP has been widely endorsed and adopted by computer manufacturers (including manufacturers of Intel Architecture systems), system software developers, and applications developers. KAP/Pro Toolset for Microsoft Windows NT provides a complete OpenMP implementation for Fortran and C developers.

The benefit/cost economies of parallel processing on two processor Intel Architecture workstations have now become quite favorable. For properly parallelized applications, one can expect substantial performance gains in return for the low cost of an additional processor and some memory. We have observed that the performance is also quite stable over a range of applications and sizes of problems, as reflected in the examples of this paper. One can view the parallelism opportunity as moving forward in time for the very economical price for additional hardware today. For example, the median speedup in this paper is above 1.7, so the effective clock speed offered by parallelism is above 1.59 GHz (1.7 x 933 MHz) (see Table 2). This scalability potential will grow as processor counts grow, multiprocessor systems become more widely available, and as successive generations of Intel Architecture workstations become available. This paper has shown that scalability results (performance speedups) of up to 98% can be gained by multithreading applications using OpenMP on Intel-Architecture based MP systems.

For more information about Intel Architecture technologies as the Intel® Pentium® III Xeon™ processor, please see the Intel web page http://www.intel.com. For specific information on Intel-based multiprocessor workstations, see http://www.intel.com/go/workstations. This site includes where-to-buy information, a cost-justification calculator for multiprocessor workstations, pointers to development tools, additional multi-processor performance information and other topics of interest to multiprocessor workstation developers, users and buyers.

9.0 OpenMP Terminology and definitions

Barrier – A thread synchronization point within the program, where all threads must reach the same point before program execution is resumed. In OpenMP, a thread memory flush is implied at the barrier, where all threads read/write thread visible variables back into memory. The barrier is a way to get all threads to a common known state.

Construct – A statement consisting of an OpenMP directive, and its corresponding structured block.

Structured block – A block of code that has only one entrance and one exit. The only exceptions to this rule are the commands that terminate execution. These are the exit() command in C/C++, and the STOP command in FORTRAN.

Master thread – The main thread of execution that creates a team of threads in parallel regions, and executes code within the master construct.

Parallel region – A region where enclosed code is executed by multiple threads. The region is identified by the user with a parallel construct, and ended with the end parallel construct.

- A region of the program that is executed by multiple threads in parallel.
- In OpenMP parallel regions may be nested within each other, but nested parallel regions are serialized by default. That is, a single thread team executes the nested parallel region.

Thread – a lightweight process.

Multiple Instruction Multiple Data (MIMD) – A parallel programming model in which multiple processors have their own set of instructions and data to process.

Single Instruction Multiple Data (SIMD) – A parallel programming model in which multiple processors execute a single set of instructions on their own set of data.

Sequential equivalence - A condition in which a program gives the expected result when executed by multiple threads as an OpenMP program or as a sequential program.

10.0 References

1 Bova, Steve, Breshears, Clay, Eigenmann, Rudolf, Gabb, Henry, Gaertner, Greg, Kuhn, Bob, Magro, Bill, Salvini, Stefano, and Vatsa, Veer. Combining Message-passing and Directives in Parallel Applications, SIAM News, November 1999, Volume 32, Number 9.

2 Kuhn, Bob and Eric Stahlberg. Porting Scientific Software to Intel® SMPs Under Windows NT.
(Part 1) **Scientific Computing & Automation** November 1997, Volume 14, Number 12: pages 31-38.
(Part 2) **Scientific Computing & Automation** January 1998, Volume 15, Number 2: pages 29-32.

3 Robichaux, Joseph, Alexander Akkerman, Curtis Bennett, Roger Jia, and Hans Leichtl. LS-DYNA 940 Parallelism on the Compaq Family of Systems, in **Proceedings of the Fifth International LS-DYNA Users' Conference**, Oct. 98, Southfield, MI.

4 Daly, Mark J. The Computational Challenge of Linkage Analysis: What Causes Disease?, **Computing in Science & Engineering**, pp. 18-32, May-June 1999.

5 Daly, Mark J., Bob Kuhn, Eamonn O'Toole, and Paul Petersen. Using SMP Parallelism with OpenMP, http://www.hgmp.mrc.ac.uk/CCP11, Issue 9, **CCP 11 Newsletter**, Oct. 99.

6 Transparent adaptive parallelism on NOWs using OpenMP; Alex Scherer, Honghui Lu, Thomas Gross, and Willy Zwaenepoel; Proceedings of the 7th ACM SIGPLAN Symposium on Principles and practice of parallel programming, 1999, Pages 96 - 106

10.1 Web Site References

- **CAChe Group, Fujitsu Systems Business of America**
  http://www.cache.fujitsu.com

- **HPVM – High Performance Virtual Machines**
  UCSD Concurrent Systems Architecture Group
  http://www-csag.ucsd.edu/projects/hpvm.html

- **InfiniBand Trade Association**
  http://www.infinibandta.org

- **Intel**
  http://www.intel.com

- **Intel – VTune Performance Analyzer**
  http://developer.intel.com/vtune/analyzer

- **Limno-Tech Inc.**
  http://www.limnotech.com

- **MPI**
  http://www.mcs.anl.gov/mpi
  http://mpi-softtech.com
  http://www.genias.de/products/patent

- **OpenMP**
  http://www.openmp.org
  http://www.kai.com

- **OpenMP Specifications**
  http://www.openmp.org/specs

- **Other parallel API's**
  http://www.roguewave.com
  http://www.sca.com/paradise.html

- **South African National Bioinformatics Institute**
  http://www.sanbi.ac.za

- U.S. Environment Protection Agency
  http://www.epa.gov

Appendix A: Test Configuration	
Processors	Dual Intel Pentium III Xeon processors 933 MHz
Chipset	Intel 840 Chipset
System Bus	133 MHz
Memory	512 MB PC800 RDRAM
Cache	256 KB L2, 16 KB L1, 16 KB instruction
Hard Disk	Seagate* ST39102LW SCSI, 7,200 RPM, 4.23 GB
Operating System	Microsoft Windows NT 4.0 1381 SP4

Appendix B: Code Sample Performing the Same Function Implemented in OpenMP, MPI, and Pthreads

OpenMP

```
#include <stdio.h>

#define N 1000

int main( int argc, char* argv[] ) {
  int i;
  float s, a[N], b[N], c[N];

  read_input();

#pragma omp parallel for reduction(+:s)
schedule(dynamic)
  for( i = 0; i < N; i++ ) {
    if( b[i] >= 0.0 ) {
      a[i] = b[i] + c[i];
      s += a[i];
    }
  }
  printf("Sum equals %d\n", s);
}
```

MPI

```
#include <stdio.h>
#include "mpi.h"

#define N 1000

int main( int argc, char* argv[] ) {
  int i;
  float s, a[N], b[N], c[N];

  /* Declare additional variables */
  int my_rank, nproc, i1, i2;
  float local_s;

  /* Initialize MPI library */
  MPI_Init( &argc, &argv );
  /* Get process id */
  MPI_Comm_Rank( MPI_COMM_WORLD,
&my_rank );
  /* Get number of MPI processes */
  MPI_Comm_Size( MPI_COMM_WORLD,
&nproc );

  /* Process zero reads input data */
  if( my_rank == 0 )
    read_input();
  /* Process zero broadcasts data to other processes
*/
  MPI_Bcast( &b, N, MPI_FLOAT, 0,
MPI_COMM_WORLD );
  MPI_Bcast( &c, N, MPI_FLOAT, 0,
MPI_COMM_WORLD );

  /* Calculate domain for each MPI process */
  i1 = my_rank * N / nproc;
  i2 = ( my_rank + 1 ) * N / nproc - 1;

  for( i = i1; i < i2; i++ ) {
    if( b[i] >= 0.0 ) {
      a[i] = b[i] + c[i];
      local_s += a[i];
    }
  }
  /* Gather local sums from all MPI processes */
  MPI_Reduce( &local_s, &s, 1, MPI_FLOAT,
MPI_SUM,
                  0, MPI_COMM_WORLD );
  MPI_Finalize();
  printf("Sum equals %d\n", s);
}
```

Pthreads

```
#include <stdio.h>
#include <pthread.h>

#define N 1000
#define NT 4

void *partial_sum( void * );

float a[N], b[N], c[N];  /* now a global variables */
float sum = 0.0;

/* Declare a mutual exclusion variable */
/* to synchronize access to global sum */
pthread_mutex_t protect_sum =
PTHREAD_MUTEX_INITIALIZER;


int main( int argc, char* argv[] ) {
  int i, my_id[NT];
  pthread_t tid[NT];  /* Thread handles */

  read_input();

  /* Create NT threads, each executing partial_sum
concurrently */
  for( i = 0; i < NT; i++ )
    my_id[i] = i;
    pthread_create( &tid[i], NULL, partial_sum,
&my_id[i] );

  /* Wait until threads are finished then rejoin to
parent thread */
  for( i = 0; i < NT; i++ )
    pthread_join( &tid[i], NULL );

  printf("Sum equals %d\n", s);
}


/* Work must be compartmentalized in functions */
void *partial_sum( void *my_id ) {
  float local_s = 0.0;
  int my_id, i1, i2;

  /* Calculate domain for each thread */
  i1 = *(int *)my_id * N / NT;
  i2 = ( *(int *)my_id + 1 ) * N / NT - 1;

  for( i = i1; i < i2; i++ ) {
    if( b[i] >= 0.0 ) {
      a[i] = b[i] + c[i];
      local_s += a[i];
    }
  }
  /* Access to the global sum must be synchronized
*/
  pthread_mutex_lock( &protect_sum);
    sum += local_s;
  pthread_mutex_unlock( &protect_sum );

  return 0;
}
```

In the **OpenMP** example code, the pragma contains all the information that the compiler needs to execute the loop in parallel. The reduction clause declares that the variable s is being used in a summation operation. The if-test could cause a load-imbalance if there is an uneven distribution of positive values in array b. Therefore, the schedule clause states that dynamic scheduling of threads should be used at runtime. The ability to change scheduling with a high-level clause is a significant advantage of OpenMP, as the MPI and Pthreads examples demonstrate.

The **MPI** version of this example code is significantly more complex. It is necessary to include an additional header file for the MPI variables, declare additional variables for the required code modifications, and the programmer must explicitly code all communication between MPI processes. The program is now inextricably linked to the MPI library. Notice that static, rather than dynamic, scheduling is used in the for-loop. With low-level methods like MPI, the programmer is also responsible for load-balancing. Dynamic load balancing would further increase the size of this program.

With **Pthreads**, an additional header and variables are also required. Significant code modifications are required because work must be compartmentalized in Pthreads programs. Specifically, the for-loop is moved to a function that can be mapped to threads using the pthread_create function. Like MPI, Pthreads is a low-level method, except rather than requiring the programmer to code interprocess communication, the programmer is now required to explicitly synchronize to variables in shared memory. This is the purpose of the pthread_mutex_lock/unlock functions. The program is now inextricably linked to the Pthreads library. Dynamic load-balancing would further complicate this program so once again static scheduling is used.

Appendix C OpenMP Fortran Version 1.0 Syntax Summary

OpenMP Fortran Version 1.0: Syntax-Summary

OpenMP is an industry-standard API for programming shared memory computers. It is based on a fork/join programming model in which a program with a single thread of execution (the master thread) spawns a team of threads to carry out work concurrently. This note is a brief summary of the OpenMP Fortran version 1.0 Application Program Interface. To learn more about OpenMP and how to use it, consult the OpenMP web site at www.openmp.org.

Directive format

```
sentinel directive-name [clause[[,]clause]…]
```

where sentinel is either fixed source:

```
!$OMP | C$OMP | *$OMP
```

or free source form

```
!$OMP
```

Without loss of generality, we will use the C$OMP sentinel throughout this summary.

Restrictions

- Fixed form sentinels must start in column 1 and have a space or zero in column 6 unless the sentinel indicates a continuation line
- Free source sentinels can appear in any column as long as it is preceded only by white space. Continuation indicated with an ampersand on the last non-blank line.

Examples

Fixed source:

```
C$OMP PARALLEL DO
C$OMP+SHARED(A, B, C)
```

Free source:

```
!$OMP PARALLEL DO &
!$OMP SHARED(A, B, C)
```

Conditional compilation

```
sentinel [legal-Fortran-line]
```

Where the sentinels vary depending on fixed source

```
!$ | C$ | *$
```

or free source

```
!$
```

During compilation, the sentinel is replaced with 2 spaces.

Parallel Region construct

```
C$OMP PARALLEL [clause[[,]clause]…]
     STRUCTURED-BLOCK
C$OMP END PARALLEL
```

Where *clause* is:

```
IF (SCALAR-LOGICAL-EXPRESSION)
PRIVATE(LIST)
FIRSTPRIVATE(LIST)
DEFAULT (PRIVATE | SHARED | NONE)
SHARED(LIST)
COPYIN(LIST)
REDUCTION({OPERATOR|INTRINSIC}: LIST)
```

Usage Note:

When the `if-clause` is present, the code within the parallel region is only executed on multiple threads if the `SCALAR-LOGICAL-EXPRESSION` evaluates to `.TRUE.` If it evaluates to `.FALSE.` the code within the parallel region is serialized; i.e. it is executed by a team of size one.

Restrictions

- The `PARALLEL/END-PARALLEL` directive pair must appear in the same routine in the executable section of the code.
- A `STRUCTURED-BLOCK` is a set of statements with a single entry at the top and a single exit at the bottom. It is illegal to branch into or out of a `STRUCTURED-BLOCK`.
- There is an implied barrier at the `END PARALLEL` construct.
- At most one `IF-CLAUSE` can appear on the directive.

Work-sharing Constructs

Work-sharing constructs and the `BARRIER` directive must be encountered in the same order by all threads in a team. They must be encountered by all threads in a team or none at all. There is an implied barrier and the end of a work-sharing construct unless a `NOWAIT` clause is specified in which case the threads immediately continue execution once they reach the end of the work-sharing construct.

DO directive

```
C$OMP DO [clause[[,] clause] …]
     do_loop
[C$OMP END DO [NOWAIT]]
```

Where *clause* is:

```
PRIVATE(LIST)
FIRSTPRIVATE(LIST)
REDUCTION({OPERATOR|INTRINSIC}: LIST)
ORDERED
SCHEDULE(TYPE[, CHUNK_SIZE])
LASTPRIVATE(LIST)
```

Usage Note:

The `ORDERED` clause tells the compiler to expect an `ORDERED` directive in the body of `the DO-LOOP`.

The SCHEDULE clause defines how the iterations are mapped onto the team of threads:

- `SCHEDULE(STATIC[, CHUNK_SIZE])`: iterations are divided into chunks of size `CHUNK_SIZE` and assigned to the members of the team in a round robin fashion. If `CHUNK_SIZE` isn't given, approximately equal sized chunks are assigned one to each thread.
- `SCHEDULE(DYNAMIC[,CHUNK_SIZE])`: iterations are divided into chunks of size `CHUNK_SIZE` and assigned one-by-one to the threads as they finish the previous chunk of iterations. When no `CHUNK_SIZE` is given, it defaults to `1.`
- `SCHEDULE(GUIDED[,CHUNK_SIZE])`: iterations are assigned to threads as with the `DYNAMIC schedule,` but the chunks are of decreasing sizes. The number of iterations in a chunk starts at some large value and decrease down to `CHUNK_SIZE.` If `CHUNK_SIZE` equals `1,` the size of each chunk is approximately the number of unassigned iterations divided by the number of threads in the team. If `CHUNK_SIZE` isn't specified, it defaults to one.
- `SCHEDULE(RUNTIME):` The schedule and chunk size are determined at runtime by setting the runtime variable `OMP_SCHEDULE.` If this variable is not set, the behavior is implementation dependent.

Restrictions:

- `CHUNK-SIZE` must be an integer or an integer expression.
- The values of the loop control parameters must be the same for all the threads in the team.
- The `DO` loop iteration variable must be of type integer.
- If used, the `END-DO` directive must appear immediately after the end of the loop.
- Only a single `SCHEDULE` clause can appear on a `DO` directive.

- Only a single `ORDERED` clause can appear on a `DO` directive.

SECTIONS/SECTION directive

```
C$OMP SECTIONS [clause[[,]clause]…]
[C$OMP SECTION]
     STRUCTURED-BLOCK
[C$OMP SECTION
     STRUCTURED-BLOCK].
C$OMP END SECTIONS [NOWAIT]
```

Where *clause* is:
```
PRIVATE(LIST)
FIRSTPRIVATE(LIST)
LASTPRIVATE(LIST)
REDUCTION({OPERATOR|INTRINSIC}: LIST)
```

Restrictions:

- A SECTION directive must not be outside the lexical extent of the SECTIONS/END SECTIONS directive pair.

SINGLE directive

```
C$OMP SINGLE [clause[[,] clause] …]
     STRUCTURED-BLOCK
C$OMP END SINGLE [NOWAIT]
```

Where *clause* is:
```
PRIVATE(LIST)
FIRSTPRIVATE(LIST)
```

Combined Parallel Work-sharing Constructs

PARALLEL DO directive

```
C$OMP PARALLEL DO [clause[[,]clause]…]
     do-loop
[C$OMP END PARALLEL DO]
```

Where *clause* is:
```
IF (SCALAR-LOGICAL-EXPRESSION)
PRIVATE(LIST)
FIRSTPRIVATE(LIST)
DEFAULT (PRIVATE | SHARED | NONE)
SHARED(LIST)
COPYIN(LIST)
SCHEDULE(TYPE[, CHUNK_SIZE])
ORDERED
LASTPRIVATE(LIST)
REDUCTION({OPERATOR|INTRINSIC}: LIST)
```

Usage Note:

The construct is the same as a PARALLEL construct immediately followed by a DO work sharing directive.

Restrictions:

This construct shares restrictions with the PARALLEL and DO directives.

PARALLEL SECTIONS construct

```
C$OMP PARALLEL SECTIONS [clause[[,]clause] … ]
[C$OMP SECTION]
     structured-block
[C$OMP SECTION
     structured-block]
C$OMP END PARALLEL SECTIONS
```

Where *clause* is:

```
IF (SCALAR-LOGICAL-EXPRESSION)
PRIVATE(LIST)
FIRSTPRIVATE(LIST)
DEFAULT (PRIVATE | SHARED | NONE)
SHARED(LIST)
COPYIN(LIST)
LASTPRIVATE(LIST)
REDUCTION({OPERATOR|INTRINSIC}: LIST)
```

Restrictions:

This construct shares restrictions with the `PARALLEL` and `SECTIONS` constructs.

Master and Synchronization Constructs

MASTER directive

```
C$OMP MASTER
     STRUCTURED-BLOCK
C$OMP END MASTER
```

ATOMIC directive

```
C$OMP ATOMIC
     expression-stmt
```

Usage Note:

The `atomic` construct is semantically equivalent to `critical` statement. The single statement *expression-stmt* must use one of the following forms:

```
x = x operator expr
x = expr operator x
x = intrinsic (x, expr)
x = intrinsic (expr, x)
```

Where *x* is a scalar variable of intrinsic type.

expr is a scalar expression that does not reference `x.`

intrinsic is one of `MAX, MIN, IAND, IOR` or `IEOR`

operator is one of `+, *, -, /, .AND, .OR., .EQV.` or `.NEQV.`

Restriction

All references to the storage location *x* are required to have the same type and type parameters.

BARRIER directive

```
C$OMP BARRIER
```

CRITICAL directive

```
C$OMP CRITICAL [(name)]
     STRUCTURED-BLOCK
C$OMP END CRITICAL [(name)]
```

Where *name* is: An identifier

FLUSH directive

```
C$OMP FLUSH [(list)]
```

Where `list` is a comma-separated list of variables that need to be flushed

A `flush` is implied by the following constructs:

- `BARRIER`
- `CRITICAL` and `END CRITICAL`
- `END DO`
- `END PARALLEL`
- `END SECTIONS`
- `END SINGLE`
- `ORDERED` and `END ORDERED`

ORDERED directive

```
C$OMP ORDERED
     Structured-block
C$OMP END ORDERED
```

Restrictions:

- An `ORDERED` directive must not be in the dynamic extent of a `DO` directive or a `PARALLEL DO` directive that does not have the `ORDERED` clause specified.
- An iteration of a loop with a DO directive must not execute the same ORDERED directive more than once, and it must not execute more than one ORDERED directive.

Data Environment Constructs and Clauses

THREADPRIVATE directive

```
C$OMP THREADPRIVATE(/cb/[,/cb/] …)
```

Where *cb* is the name of the common block to be made `private` to a thread.

Restrictions

- The `THREADPRIVATE` directive must appear after every declaration of a thread private common block.
- Only named common blocks can be made thread private.
- It is illegal for a `THREADPRIVATE` common block or its constituent variables to appear in any clause other than a `COPYIN` clause. They are not affected by the `DEFAULT` clause.

COPYIN clause

COPYIN (*list*)

where *list* contains `TRHEADPRIVATE` common blocks or variables included in a `TRHEADPRIVATE` common block.

DEFAULT clause

DEFAULT(PRIVATE | SHARED | NONE)

Restrictions:

Only a single default clause may be specified on a parallel directive.

FIRSTPRIVATE clause
FIRSTPRIVATE(list)
LASTPRIVATE clause
LASTPRIVATE(list)
PRIVATE clause
PRIVATE(list)

Restrictions:
Variables that are specified PRIVATE on a PARALLEL directive cannot be specified PRIVATE again on an enclosed work-sharing directive. As a result, variables that are specified PRIVATE on a work-sharing directive must be SHARED in the enclosing parallel region.

REDUCTION clause
REDUCTION ({operator | intrinsic}:*list*)
Where variables in *list* are scalar and of intrinsic type and *operator* or *intrinsic* are one of:

+	Initial value = 0
*	Initial value = 1
-	Initial value = 0
.AND.	Initial value = .TRUE.
.OR.	Initial value = .FALSE.
.EQV.	Initial value = .TRUE.
.NEQV.	Initial value = .FALSE.
MAX	Initial value = Smallest representable number
MIN	Initial value = Largest representable number
IAND	Initial value = All bits on
IOR	Initial value = 0
IEOR	Initial value = 0

Usage Note:
A reduction is typically used in a statement with one of the following forms:
x = x operator expr
x = expr op x (*except for subtraction*)
x = intrinsic (x, expr)
x = intrinsic (expr, x)
IF (X .LT. expr) X = expr
Restrictions:
expr ca3999n not reference x.
Variables that appear in a REDUCTION clause must be SHARED in the enclosing context.
SHARED clause
SHARED(list)

Data Environment Rules

An OpenMP Fortran program must adhere to the following rules and restrictions with respect to data scope:

- Sequential DO loop control variables in the lexical extent of a PARALLEL region that would otherwise be SHARED based on default rules, are automatically made private on the PARALLEL directive.
- Variables that are privatized in a parallel region cannot be privatized again in an enclosed work-sharing directive. As a result, variables that appear in the PRIVATE, FIRSTPRIVATE, LASTPRIVATE, and REDUCTION clauses on a work-sharing directive must have shared scope in the enclosing parallel region.
- Assumed-size and assumed-shape arrays cannot be specified as PRIVATE, FIRSTPRIVATE, or LASTPRIVATE.
- Fortran pointers and allocatable arrays can be declared as PRIVATE or SHARED but not as FIRSTPRIVATE or LASTPRIVATE.
- Within a parallel region, the initial status of a PRIVATE pointer is undefined.
- Scope clauses apply only to variables in the static extent of the directive on which the clause appears, with the exception of variables passed as actual arguments. Local variables in called routines that don't have the SAVE attribute are PRIVATE. Common blocks and modules in called routines in the dynamic extent of a parallel region always have an implicit SHARED attribute, unless they are THREADPRIVATE common blocks.

- When a named common block is declared as PRIVATE, FIRSTPRIVATE or LASTPRIVATE, none of its constituent elements may be declared in another scope attribute. When individual members of a common block are privatized, the storage of the specified variables is no longer associated with the storage of the common block itself.
- Variables that are not allowed in the PRIVATE and SHARED clauses are not affected by the DEFAULT(PRIVATE) or DEFAULT(SHARED) clauses.
- Clauses can be repeated as needed, but each variable can appear explicitly in only one clause per directive, with the following exceptions: (1) a variable can be specified as both FIRSTPRIVATE and LASTPRIVATE; (2) Variables affected by the DEFAULT clause can be listed explicitly in a clause to override the default specification.

Directive binding

An OpenMP Fortran program must adhere to the following rules with respect to directive binding:

- The DO, SECTIONS, SINGLE, MASTER, and BARRIER directives bind to the dynamically enclosing PARALLEL, if one exists.
- The ORDERED directive binds to the dynamically enclosing DO.
- ATOMIC and CRITICAL directives enforce access with respect to ATOMIC and CRITICAL directives in all threads, not just the current team.
- A directive can never bind to any directive outside the closest enclosing PARALLEL.

Directive Nesting

An OpenMP Fortran program must adhere to the following rules with respect to the dynamic nesting of directives:

- A PARALLEL directive dynamically inside another PARALLEL directive logically establishes a new team, which is composed of only the current thread unless nested parallelism is enabled.
- DO, SECTIONS, and SINGLE directives that bind to the same PARALLEL directive are not allowed to be nested one inside the other. Furthermore, these directives are not allowed in the dynamic extent of CRITICAL and MASTER directives.
- BARRIER directives are not permitted in the dynamic extent of DO, SECTIONS, SINGLE, MASTER and CRITICAL directives
- MASTER directives are not permitted in the dynamic extent of DO, SECTIONS, and SINGLE, directives.
- ORDERED sections are not allowed in the dynamic extent of CRITICAL sections.
- Any directive set that is legal when executed dynamically inside a PARALLEL region is also legal when executed outside a PARALLEL region. When executed dynamically outside a user-specified parallel region, the directive is executed with respect to a team composed of only the master thread.

Runtime Library Functions

In the description of these routines, scalar_integer_expr is a default scalar integer expression, scalar_logical_expr is a default scalar logical expression, and var is of type INTEGER and a KIND large enough to hold an address.

Execution environment functions

SUBROUTINE OMP_SET_NUM_THREADS(scalar_integer_expr)
INTEGER FUNCTION OMP_GET_NUM_THREADS()
INTEGER FUNCTION OMP_GET_MAX_THREADS()
INTEGER FUNCTION OMP_GET_THREAD_NUM()
INTEGER FUNCTION OMP_GET_NUM_PROCS()
LOGICAL FUNCTION OMP_IN_PARALLEL()
SUBROUTINE OMP_SET_DYNAMIC(scalar_logical_expr)
LOGICAL FUNCTION OMP_GET_DYNAMIC()
SUBROUTINE OMP_SET_NESTED (scalar_logical_expr)
LOGICAL FUNCTION OMP_GET_NESTED()

Lock functions

SUBROUTINE OMP_INIT_LOCK (var)
SUBROUTINE OMP_DESTROY_LOCK(var)
SUBROUTINE OMP_SET_LOCK(var)
SUBROUTINE OMP_UNSET_LOCK(var)
LOGICAL FUNCTION OMP_TEST_LOCK (var)

Environment Variables

OMP_SCHEDULE "schedule[, chunk_size]"
OMP_NUM_THREADS int
OMP_DYNAMIC TRUE || FALSE

OMP_NESTED TRUE || FALSE

Appendix D: OpenMP C/C++ Version 1.0: Syntax Summary

OpenMP C/C++ Version 1.0: Syntax-Summary

OpenMP is an industry-standard API for programming shared memory computers. It is based on a fork/join programming model in which a program with a single thread of execution (the master thread) spawns a team of threads to carry out work concurrently. This note is a brief summary of the OpenMP C/C++ version 1.0 Application Program Interface. To learn more about OpenMP and how to use it, consult the OpenMP web site at www.openmp.org.

Directive format

#pragma omp directive-name [clause[clause]…]

The directive applies to at most one succeeding statement which must be a structured block. A structured block is a block of one or more statements with a single point of entry at the top and a single exit at the bottom. Branches into or out of a structure block are not permitted. It is allowed to have an `exit()` statement within a structured block.

Conditional compilation with the OpenMP macro

Conditional compilation is specified with the standard C/C++ preprocessor and the `_OPENMP` macro:

```
#ifdef _OPENMP
        any legal C/C++ constructs
#endif
```

The macro must not be the subject of a `#define` or an `#undef`.

Parallel region construct

#pragma omp parallel [*clause*[*clause*] …]

        structured-block

Where *`clause`* is:

- if (scalar-expression)
- private(list)
- firstprivate(list)
- default (shared | none)
- shared(list)
- copyin(list)
- reduction(operator: list)

Usage Note:

When the `if-clause` is present, the code within the parallel region is only executed with multiple threads if the `scalar-expression` evaluates to a non-zero value. If it evaluates to `zero` the code within the parallel region is serialized; i.e. it is executed by a team of size one.

Restrictions

- At most one `if clause` can appear on the directive.
- It is unspecified whether any side-effects inside the `if expression` occur.
- A throw in C++ from a parallel region must not cross a structured block, and it must be caught by the same thread that threw the exception.
- There is an implied barrier at the end of a parallel region.

Work-sharing Constructs

Work-sharing constructs and the `barrier` directive must be encountered in the same order by all threads in a team. They must be encountered by all threads in a team or none at all. There is an implied barrier and the end of a work-sharing construct unless a `nowait` clause is specified in which case the threads immediately continue execution.

for construct

#pragma omp for [*clause*[clause] …]
for-loop

Where `clause` is:
private(list)
firstprivate(list)
reduction(operator: list)
ordered
schedule(kind[, chunk_size])
nowait
lastprivate(list)

Usage Note:

The `for` loop must have the standard form
for(init-expr; var logical-op b; incr-expr)
where

- `init-expr` is a simple assignment to an integer type.
- `var` is a signed integer variable. It is implicitly made private if it has not been explicitly made private by the programmer. This variable must not be modified within the loop.
- `Logical-op` must be one of `<, <=, >,` or `>=`
- `Incr-expr` must a decrement or increment op applied to `var`, or a simple increment/decrement assignment operation with a loop invariant integer expression.
- `lb`, `b` and `incr` are loop invariant integer expressions. No synchronization takes place during the evaluation of these expressions and any side effects produce indeterminate results.
- The `ordered` clause tells the compiler to expect an `ordered` directive in the body of the `for` loop. The `schedule` clause defines how the iterations are mapped onto the team of threads:
- schedule(static[, chunk_size]): iterations are divided into chunks of size chunk_size and assigned to the members of the team in a round robin fashion. If chunk_size isn't given, approximately equal sized chunks are assigned one to each thread.
- schedule(dynamic[, chunk_size]): iterations are divided into chunks of size chunk_size and assigned one-by-one to the threads as they finish the previous chunk of iterations. When no chunk_size is given, it defaults to 1.
- schedule(guided[, chunk_size]): iterations are assigned to threads as with the dynamic schedule, but the chunks are of decreasing sizes. The number of iterations in a chunk start at some large value and decrease down to chunk_size. If chunk_size equals 1, the size of each chunk is approximately the number of unassigned iterations divided by the number of threads in the team. If chunk_size isn't specified, it defaults to one.
- schedule(runtime): The schedule and chunk size are determined at runtime by setting the runtime variable OMP_SCHEDULE. If this variable is not set, the behavior is implementation dependent.

Restrictions:

- The `for` loop must be a structured block. Its execution can not be terminated by a `break` statement.
- The values of the loop control expressions in the `for` loop associated with a `for` directive must be the same for all the threads in the team.
- The `for` loop iteration variable must have a signed integer type.
- Only a single `schedule` clause can appear on a `for` directive.
- Only a single `ordered` clause can appear on a `for` directive.
- It is unspecified if or how often any side effects within the `chunk_size`, `lb`, `b`, or `incr` expressions occur.
- The value of the `chunk_size` expression must be the same for all threads in the team.

sections/section construct

#pragma omp sections [*clause*[*clause*] …]
#pragma omp section

Where `clause` is:
private(list)
firstprivate(list)
lastprivate(list)

reduction(operator: list)

Usage Note:
The sections construct is used as follows with a sequence of structured blocks
#pragma omp sections [*clause*[*clause*] …]
{
[#pragma omp section]
structured-block
[#pragma omp section

]
}

Restrictions:
- A `section` directive must be inside the lexical extent of a `sections` directive.

single construct
#pragma omp single [*clause*[*clause*] …]

Where `clause` is: private(list)
firstprivate(list)
nowait

Combined Parallel Work-sharing Constructs

parallel for construct
#pragma omp parallel for [*clause*[*clause*] …]
for-loop

Where `clause` is: if (scalar-expression)
private(list)
firstprivate(list)
default (shared | none)
shared(list)
copyin(list)
schedule(type[, chunk_size])
ordered
nowait
lastprivate(list)
reduction(operator: list)

Usage Note:
The construct is the same as a `parallel` construct immediately followed by a `for` work sharing construct.

Restrictions:
This construct shares restrictions with the `parallel` and `for` constructs.

parallel sections construct
#pragma omp parallel sections [*clause*[*clause*] …]
#pragma omp section

Where `clause` is: if (scalar-expression)
private(list)
firstprivate(list)

default (shared | none)
shared(list)
copyin(list)
lastprivate(list)
reduction(operator: list)

Usage Note:
This construct is the same as a `parallel` construct followed by a `sections` directive.

Restrictions:
This construct shares restrictions with the `parallel` and `sections` constructs.

Master and Synchronization Constructs

master construct
#pragma omp master

atomic construct
#pragma omp atomic
expression-stmt

Usage Note:
The `atomic` construct is semantically equivalent to `critical` statement. The single statement `expression-stmt` must use one of the following forms:
#pragma omp atomic
x binop = expr or x++ or ++x or x-- or --x

Where x is an lvalue expression of scalar type and no side effects.
expr is an lvalue expression with no side effects. It must not reference x.
binop is not overloaded and is one of +, *, -, /, &, ^, <<, >>, |

Restriction
All atomic references to the storage location x throughout the program are required to have a compatible type.

barrier construct
#pragma omp barrier

Restrictions:
The smallest statement that contains a `barrier` directive must be a block (or a compound statement).

critical construct
#pragma omp critical [(*name*)]

Where *`name`* is: An identifier

flush construct
#pragma omp flush [(*list*)]
Where *list* is a comma-separated list of variables that need to be flushed
A `flush` is implied by the following constructs:

- `barrier`
- At entry to and exit from `critical`
- At entry to and exit from `ordered`

- At the exit from `parallel`.
- At exit from `for`
- At exit from `sections`
- At exit from `single`

Restriction
A variable specified in a `flush` must not have a reference type.

ordered construct
#pragma omp ordered
Restrictions:
- An `ordered` directive can only appear in the dynamic extent of a `for` directive that has the `ordered` clause specified.
- An iteration of a loop with a `for` construct must not execute the same `ordered` directive more than once, and it must not execute more than one `ordered` directive.

Data Environment Constructs and Clauses

threadprivate construct
#pragma omp threadprivate(list)
Where list is a comma separated list of variables that do not have an incomplete type.

Restrictions
- A `threadprivate` variable must not appear in any clause other than the `copyin`, schedule, or the if clause. A default clause does not effect a `threadprivate` variable.
- The address of a `threadprivate` variable is not an address constant.
- A `threadprivate` variable must not have a reference type.
- A `threadprivate` variable with class type must have an accessible, unambiguous default constructor.

copyin clause
copyin (list)
where `list` contains `threadprivate` variables.

Restrictions:
- A variable that is specified in a `copyin` clause must have an accessible, unambiguous copy assignment operator.

default clause
default(shared | none)

Restrictions:
- Only a single `default` clause may be specified on a `parallel` directive.

firstprivate clause
firstprivate(list)

Restrictions:
- All restrictions on `private` apply except for the restriction on `const`-qualified types.
- A variable with a class type that is specified `firstprivate` must have an accessible unambiguous copy constructor.

lastprivate clause
lastprivate(list)

Restrictions:
All restrictions on `private` apply.
A variable that is specified as `lastprivate` must have an accessible, unambiguous copy assignment operator.

private clause
private(list)

Restrictions:

- A variable with a class type that is specified as `private` must have an accessible, unambiguous default constructor.
- Unless it has a class type with a mutable member, a variable specified as `private` must not have a `const`-qualified type.
- A variable specified as `private` must not have an incomplete type or a reference type.
- Variables that are specified `private` on a `parallel` directive cannot be specified `private` again on an enclosed work-sharing or `parallel` directive. As a result, variables that are specified `private` on a work-sharing or `parallel` directive must be specified `shared` in the enclosing parallel region

reduction clause
reduction (*op*:list)

Where *op* is:	+	Initial value = 0
	*	Initial value = 1
	-	Initial value = 0
	&	Initial value = ~0
	|	Initial value = 0
	^	Initial value = 0
	&&	Initial value = 1
	||	Initial value = 0

Usage Note:
A reduction is typically used in a statement with one of the following forms:
x = x op expr
x <op> = expr
x = expr op x (except for subtraction)
x++ or ++x or x-- or --x
where `expr` does not reference `x`.

Restrictions:

- The type of the variables in the `reduction` clause must be valid for the `reduction operator` except that pointer types and reference types are never permitted.
- A variable that is specified in the `reduction` clause must not be `const`-qualified.
- A variable that is specified in the `reduction` clause must be `shared` in the enclosing parallel region.

shared clause
shared(list)

Directive binding
An OpenMP C/C++ program must adhere to the following rules with respect to directive binding:

- The `for`, `sections`, `single`, `master` and `barrier` directives bind to the dynamically enclosing `parallel`, if one exists. If no parallel region is currently being executed, the directives apply to a team consisting of the master thread.
- The `ordered` directive binds to the dynamically enclosing `for`.
- The `atomic` directive enforces exclusive access with respect to `atomic` directives in all threads, not just the current team.
- The `critical` directive enforces exclusive access with respect to `critical` directives in all threads, not just the current team.
- A directive can never bind to any directive outside the closest enclosing `parallel`.

Directive Nesting
An OpenMP C/C++ program must adhere to the following rules with respect to the dynamic nesting of directives:

- A `parallel` directive dynamically inside another `parallel` logically establishes a new team which is composed of only the current thread, unless nested parallelism is enabled.
- `For`, `sections`, and `single` directives that bind to the same `parallel` are not allowed to be nested inside each other.

- `Critical` directives with the same name are not allowed to be nested inside each other
- `For`, `sections` and `single` directives are not permitted in the dynamic extent of `critical`, `ordered` and `master` regions.
- `Barrier` directives are not permitted in the dynamic extent of `for`, `ordered`, `sections`, `single`, `master` and `critical` regions.
- `Master` directives are not permitted in the dynamic extent of `for`, `sections`, and `single` directives.
- `Ordered` directives are not allowed in the dynamic extent of `critical` regions.
- Any directive that is permitted when executed dynamically inside a parallel region is also permitted when executed outside a parallel region.. When executed dynamically outside a user-specified parallel region, the directive is executed with respect to a team composed of only the master thread.

Runtime Library Functions

These routines use the include file <omp.h>. This file includes function prototypes and defines the type `omp_lock_t`.

Execution environment functions

```
void omp_set_num_threads(int num_threads);
int omp_get_num_threads(void);
int omp_get_max_threads(void);
int omp_get_thread_num(void);
int omp_get_num_procs(void);
int omp_in_parallel(void);
void omp_set_dynamic(int dynamic_threads);
int omp_get_dynamic(void);
void omp_set_nested(int nested);
int omp_get_nested(void);
```

Lock functions

```
void omp_init_lock(omp_lock_t *lock);
void omp_init_nest_lock(omp_nest_lock_t *lock);

void omp_destroy_lock(omp_lock_t *lock);
void omp_destroy_nest_lock(omp_nest_lock_t *lock);

void omp_set_lock(omp_lock_t *lock);
void omp_set_nest_lock(omp_nest_lock_t *lock);

void omp_unset_lock(omp_lock_t *lock);
void omp_nuset_nest_lock(omp_nest_lock_t *lock);

int omp_test_lock(omp_lock_t *lock);
int omp_test_nest_lock(omp_nest_lock_t *lock);
```

Environment Variables

```
OMP_SCHEDULE "schedule[, chunk_size]"
OMP_NUM_THREADS int
OMP_DYNAMIC TRUE || FALSE
OMP_NESTED TRUE || FALSE
```

About these documents

These appendix sections C,D were written by Tim Mattson of Intel Corporation. They are based on the OpenMP specifications "OpenMP Fortran Applications Program Interface" version 1.0 dated October 1997 and "OpenMP C/C++ Applications Program Interface" version 1.0 dated October 1998 respectively

---

Performance Improvements on Intel Architecture based Multiprocessor Workstations: Multithreaded Applications using OpenMP

Disclaimers and Restrictions

Performance Report Notice

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of
Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.
Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, reference:
www.intel.com/procs/perf/limits.htm or call (U.S.) 1-800-628-8686 or 1-916-356-3104.